

Data Structures and Algorithms

**Lab 4: DSA\_Lab4.h**

**The Scenario**

Last lab you implemented Doubly-Linked Lists (Your sword) along with Iterators (Your shield). With your newly-created sword and shield, you ride your stallion out to face the mighty dragon. It is, however, one thing to **create** a sword and shield and another to **use** them. As you make camp for the night, you begin to train. In today’s lab, you will be using the C++ **list** data structure to learn more about how to use lists and how to program for them.

When working with lists, there are several advantages and disadvantages compared to arrays. To create efficient programs for both the memory and the processor, it’s important to be able to program with the strengths of each. At the end of the lab is a table demonstrating some of the advantages and disadvantages of each.

One final thing before you begin your training! When storing data within lists, you have many options on how to organize, order, add, and remove data. Two of the most common ways are using **queue ordering** and **stack ordering**. Queue ordering runs in **First In First Out** (**FIFO**) order. Think of it like a line to get into a Black Sabbath reunion concert. The first people to get in line are the first leave the line when the doors open to find your seats. Stacks, however, run in **First In Last Out** (**FILO**) order. This is more like a stack of pancakes where the last pancake to be put on the stack is on the top, and thus, the first to be taken off the stack and eaten.

Some real-world examples of each are the Call Stack in programming—the last call to a recursive function is the first to be evaluated. An example of a queue is a printer queue, where the first document sent to the printer is the first to be printed.

**What To Do…**

Open DSA\_Lab4.h. There will be instructions written in the comments on what is expected. Below is the gist of each function and variable.

***Variables:***

**mList** The only variable explicitly needed today! The list you will be using to store data, add and subtract data from, and do most of your data manipulation.

***Functions:***

**QueueOrderingAdd** Accepts an array and a size. The size variable represents the size of the array and thus the number of elements to add to the list. (As the name implies, the elements must be added in using Queue Ordering.) The list must be cleared out and be at size 0 before adding elements here.

**StackOrderingAdd** Accepts an array and a size. This version uses Stack Ordering.

**QueueOrderingRemove** Removes a single element from the list using queue ordering.

**StackOrderingRemove** Removes a single element from the list using Stack ordering.

**Insert (index)** Insert a value right before the node at the index. See code comments for examples. (Can be used to add a new head.)

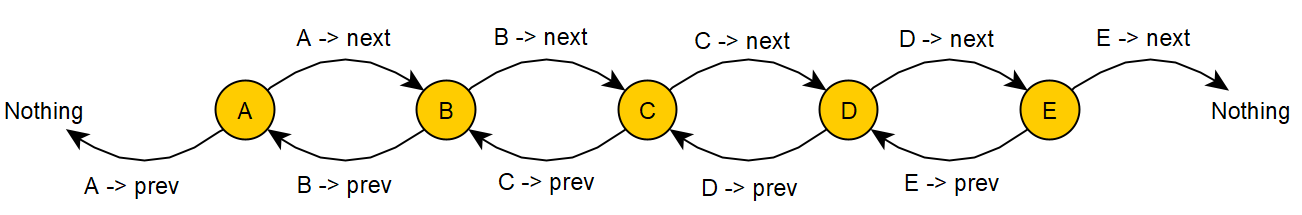
**Insert (iterator)** This time an iterator is passed in “bookmarking” the spot to insert. Insert the element passed in right before it.

**RemoveDecimalGreater** Checks each element of the list and removes it if the value after the decimal is greater than the value passed in. It can be assumed that the value passed in will be between 0.0 and 1.0.

**ex. 0.5** is passed in. **25.7** is in the list. It is **greater** than 0.5 in this case and should be removed. **25.4** is also in the list. It is considered less than 0.5 and is not removed.

**Tips, Tricks, and Resources**

* Functions/Data Members available in the List class ( i.e. myList.pop\_back( ) ) can be found on the Cplusplus.com documentation for lists:
  + <http://www.cplusplus.com/reference/vector/vector/>
* In case you want to use it to visualize what’s going on under-the-hood in C++’s list class, here is the illustration of DLists from last lab’s handout, though you won’t need to manipulate the pointers this lab:



See next page.

* **A note on Arrays, Lists, and some of the advantages/disadvantages of each:**

|  |  |
| --- | --- |
| **Arrays** | **Lists** |
| * Allow for **random access** | * Must iterate to the element needed. |
| * Requires all elements to be stored next to each other in memory | * Elements can be stored anywhere in memory |
| * Changing the size involves creating an entirely new array and copying the data over. | * Changing the size is as simple as adding and removing elements. |
| * Generally **more** processor-efficient | * Generally **less** processor-efficient |
| * Generally **less** memory-efficient | * Generally **more** memory-efficient |
| * Easier to use | * Harder to use |

Many detriments of both Arrays and Lists can be mitigated through adding supplementary functionality, as we’ve seen in previous labs. For example, creating and using dynamic arrays (Vectors) gives arrays list-like functionality. In another example, creating Iterators allows us to use lists much in the way we use Arrays and even gives us array-like random access. However, at their hearts, the Dynamic Arrays and Lists with Iterators suffer from the memory and processor draw-backs associated with their more basic counterparts.

**Plagiarism**

Plagiarism and Academic Dishonesty are considered a **very** serious offense in this class and can have a range of consequences including suspension, and in very serious cases, expulsion. If you either share your code or copy someone else’s code, you will be given a **0** on your lab and can face further disciplinary action.

In other words, don’t cheat please!